

# Enterprise Architektur-Muster

JULIAN BRUDER\*, ABDELLAH FILALI\*, and LUCA FRANKE\*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

Blah abstrakt...

## 1 EINLEITUNG

Mit E-Commerce-Beispiel motivieren

## 2 GRUNDLAGEN VON ENTERPRISE-ARCHITEKTUREN

Verteilte Systeme ...

Architekturen ...

Komponenten ...

...

## 3 KLASSISCHE ENTERPRISE-ARCHITEKTUREN

### 3.1 Service-oriented

Die bisherigen Architekturmuster, wie die Monolithic Architecture waren durch ihren engen gekoppelten Funktionalitäten charakterisiert. Jetzt betrachten wir die Service-oriented Architecture (SOA), die als zentrales Konzept von Dienste (englisch *Services*) betont. Ein Dienst kapselt eine Funktionalität und stellt diese über eine fest definierte Schnittstelle bereit [Bianco et al. 2007][S. 3].

Eine universelle Definition von SOA ist nicht vorhanden, jedoch ist das Konzept von Dienst ein zentrales Element der Architektur. Ein Dienst besitzt folgenden Eigenschaften [Bianco et al. 2007, 16]:

- Der Dienst ist eine unabhängige Einheit
- Der Dienst ist über einen Namen oder eine Adresse erreichbar über das Netzwerk
- Der Dienst verfügt über eine publizierte Schnittstelle
- Der Dienst ist durch eine Registry auffindbar
- Der Dienst kann zur Laufzeit dynamisch lokalisiert und genutzt werden

Betrachten wir im Folgenden die Hauptbestandteile der SOA, wie die Abb. 1 [Endrei et al. 2004] darstellt:

- Service Provider: Bietet einen spezifischen Dienst an
- Service Bus: Dienst, das die Orchestrierung der Kommunikation zwischen Consumer und Provider gewährleistet
- Service Consumer: Nutzt einen bereitgestellten Dienst, dieses kann einen End-User oder ein anderer Dienst sein
- Service Registry: Dient als zentrale Sammlung von Meta-Daten über die verfügbare Dienste

Ein Service Provider registriert die Meta-Daten seines Diensts in den Service Registry. Dieses ermöglicht die dynamische Entdeckung der Dienste zur Laufzeit. Die gespeicherte Meta-Daten bestehen aus der Interface-Vertrag und der Endpoint des Providers. Der Endpoint stellt die Schnittstelle zum Dienst dar, während der Interface-Vertrag des Service Provider festlegt, wie der Consumer den Dienst nutzen kann. [Endrei et al. 2004]

\*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt. Diese Arbeit ist unter der Lizenz ??? freigegeben.

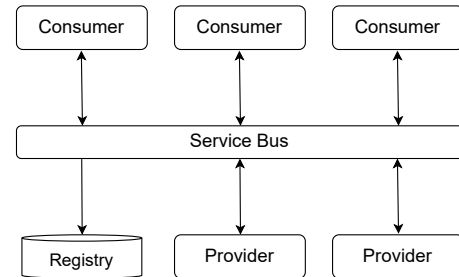


Abb. 1. Aufbau der Service-oriented Architecture

Der Service Consumer kann dann über den Service Bus den benötigten Dienst im Registry entdecken und gemäß dem klar definierten Vertrag nutzen.

Ein Service Bus, insbesondere ein Enterprise Service Bus (ESB), übernimmt die zentrale Aufgabe der Orchestrierung der Kommunikation zwischen Provider und Consumer. Zu den typischen Funktionen eines ESB gehören u.a. das Routing, sowie die Konvertierung der Nachrichten, wenn Dienste unterschiedliche Protokolle für die Kommunikation verwenden.

Für kleine Unternehmen, die nur über eine begrenzte Anzahl an Anwendungen und Services kann der Einsatz von ESB jedoch zu aufwendig sein. In solche Fällen kann eine leichtgewichtige Point-to-Point-Kommunikation sinnvoll sein. [Endrei et al. 2004, 21] Dabei wird die direkte Verbindung zwischen Consumer und Provider individuell entworfen, implementiert und gewartet. [Endrei et al. 2004, 19]

Da jeder Service einen spezifischen Dienst anbieten und diese durch eine klar definierte Schnittstellen, können Dienste unabhängig voneinander weiterentwickelt werden, was zu einer höheren Agilität führt.

Dienste können in anderen Anwendungen erneut eingesetzt oder mit anderen Diensten kombiniert werden, um die Bedürfnisse eines Unternehmens zu erfüllen. Dies führt zu weniger Duplikaten von Ressourcen, einer höheren Wiederverwendbarkeit von Code und reduzierten Kosten [Endrei et al. 2004].

Betrachten wir erneut das E-Commerce-Beispiel aus der Einleitung. Dafür definieren wir drei Arten von Services:

- OrderService: Ein Dienst, das den gesamten Bestellvorgang initiiert
- PaymentService: Ein Dienst, das den Bezahlvorgang darstellt
- ShipmentService: Ein Dienst, das den Versandvorgang darstellt

Wie Abbildung 2 zeigt, die Dienste ShipmentService, PaymentService und OrderService registrieren ihre Dienste bei dem ServiceRegistry. Die Einträge im ServiceRegistry bestehen aus einem Identifikator,

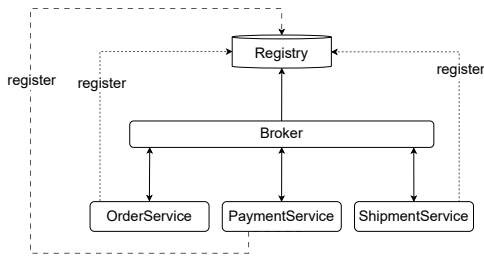


Abb. 2. E-Commerce-Beispiel mit Service-oriented Architecture

eine Beschreibung der Input und Output der Anfrage an dem Dienst, sowie den Endpoint für den Zugriff der Dienste.

In diesem Beispiel OrderService konsumiert die beiden anderen Dienste. Dieses erfolgt durch Anfragen über den Broker, der die ServiceRegistry anspricht. Nach Erhalt der Meta-Daten, der OrderService Kann die benötigten Dienste gemäß den Interface-Verträge nutzen.

Sollte sich die Implementierung oder die Schnittstellen der Provider ändern, ist der Consumer nicht davon betroffen, da die Dienste in der ServiceRegistry dynamisch zu Laufzeit lokalisieren und genutzt werden.

Außerdem jeder Komponente dieser Architektur ist lose gekoppelt, was die Austauschbarkeit der einzelnen Komponenten erleichtert.

Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub<sup>1</sup> zu finden.

## 4 MODERNE ENTERPRISE-ARCHITEKTUREN

### 4.1 Event-Driven Architecture

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier „An introduction to Software Architecture“ auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert

- Konsument (englisch *Consumer*): Komponente, die auf publizierte Events reagiert
- Vermittler (englisch *Mediator*): Komponente zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 3 stellt diesen Vertrag dar.

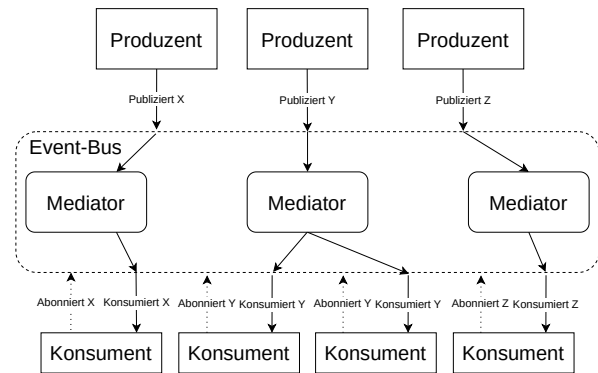


Abb. 3. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht agilen Entwicklungsteams durch klar abgegrenzte Features einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der locken Kopplung maximale Skalierung. Daher sind Event-Driven Architekturen besonders für datenintensive Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Betrachten wir erneut das E-Commerce-Beispiel aus der Einleitung. Dafür definieren wir drei Arten von Events:

- OrderCreated: Ein Event, das genau dann erzeugt wird, wenn eine neue Bestellung aufgegeben wird
- PaymentProcessed: Ein Event, das genau dann erzeugt wird, wenn der Bezahlvorgang abgeschlossen wird
- ShipmentInitiated: Ein Event, das genau dann erzeugt wird, wenn die Bestellung versandt wird

Weiter teilen wir die Funktionalität ähnlich wie bei der Microservice-Architektur in die drei verschiedenen Dienste OrderService, PaymentService und ShipmentService auf.

<sup>1</sup><https://github.com/Beleg-6-EAP/demo-soa-ecommerce>

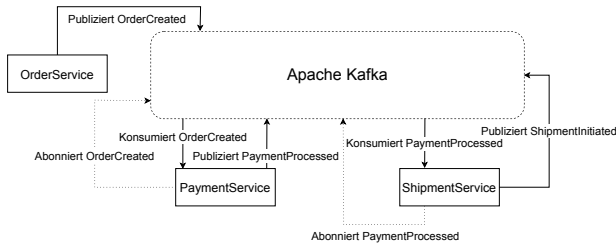


Abb. 4. E-Commerce-Beispiel mit Event-Driven Architecture

Wie Abbildung 4 zeigt, sind alle drei Dienste Produzenten und Publisher, erzeugen also Events und veröffentlichen diese. Die Dienste PaymentService und ShipmentService sind zudem Konsumenten, sodass ersterer auf Events des Typs OrderCreated und zweiterer auf Events des Typs ShipmentInitiated reagiert. Eine beispielhafte Implementierung des PaymentService mit Apache Kafka als Event-Broker ist im Anhang A.1 zu finden. Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub<sup>2</sup> zu finden.

Das Beispiel zeigt, dass die Event-Driven Architektur mit weiteren agilen Strukturen wie Microservices kombiniert werden kann, was die Agilität der Architektur weiter erhöht. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

## 5 FALLSTUDIEN UND PRAXISBEISPIELE

Blah ...

## 6 DISKUSSION

## 7 ZUSAMMENFASSUNG UND AUSBLICK

### LITERATUR

- Phil Bianco, Rick Kotermanski, and Paulo Merson. 2007. *Evaluating a service-oriented architecture*. Citeseer.
- Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. 2004. *Patterns: service-oriented architecture and web services*. IBM Corporation, International Technical Support Organization New York, NY ....
- David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture*. Technical Report CMU/SEI-94-TR-021. <https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/> Accessed: 2025-Jan-2.
- Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. <https://doi.org/10.21275/SR24820051042>
- Thomas Michl. 2018. *Das agile Manifest – eine Einführung*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. [https://doi.org/10.1007/978-3-662-57699-1\\_1](https://doi.org/10.1007/978-3-662-57699-1_1)
- Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. <https://doi.org/10.1016/j.iot.2023.100854>

<sup>2</sup><https://github.com/Beleg-6-EAP/demo-eda-ecommerce>

## A CODE-BEISPIELE

### A.1 Event-Driven-Architecture

```

1 package demo.eda.service;
2
3 import demo.eda.event.OrderCreatedEvent;
4 import demo.eda.event.PaymentProcessedEvent;
5 import demo.eda.model.Payment;
6 import demo.eda.repository.PaymentRepository;
7 import lombok.RequiredArgsConstructor;
8 import org.springframework.kafka.annotation.KafkaListener;
9 import org.springframework.kafka.core.reactive.ReactiveKafkaProducerTemplate;
10 import org.springframework.stereotype.Service;
11 import reactor.core.publisher.Flux;
12 import reactor.kafka.receiver.KafkaReceiver;
13 import reactor.kafka.receiver.ReceiverOptions;
14
15 import java.util.UUID;
16
17 @Service
18 @RequiredArgsConstructor
19 public class PaymentService {
20
21     private final PaymentRepository paymentRepository;
22     private final ReactiveKafkaProducerTemplate<String, PaymentProcessedEvent> producer;
23
24     @KafkaListener(topics = OrderCreatedEvent.TOPIC, groupId = "payment-service")
25     public Flux<Void> processPayment(ReceiverOptions<String, OrderCreatedEvent> receiverOptions) {
26         return KafkaReceiver.create(receiverOptions)
27             .receive()
28             .flatMap(record -> {
29                 final OrderCreatedEvent event = record.value();
30                 final Payment payment = new Payment(UUID.randomUUID().toString(), event.getOrderID(), true);
31                 return paymentRepository.save(payment).flatMap(savedPayment -> {
32                     final PaymentProcessedEvent paymentEvent =
33                         new PaymentProcessedEvent(
34                             savedPayment.getOrderID(),
35                             savedPayment.getId(),
36                             savedPayment.isSuccess());
37                     return producer.send(PaymentProcessedEvent.TOPIC, paymentEvent).then();
38                 });
39             });
40     }
41 }

```

Listing 1. Service-Implementierung des PaymentService in Java Spring Boot 3.4.1 mit Apache Kafka als Event-Broker

## B ÜBUNGSAUFGABEN

### B.1 Übungsaufgabe 1

Blah ...